下载APP

08｜内核跟踪（下）：开发内核跟踪程序的进阶方法

2022-02-02 倪朋飞

《eBPF 核心技术与实战》 课程介绍 >

讲述：山荣
时长 22:18 大小 20.43M

你好，我是倪朋飞。

上一讲，我带你梳理了查询 eBPF 跟踪点的常用方法，并以短时进程的跟踪为例，通过 bpftrace 实现了内核跟踪点的跟踪程序。

bpftrace 简单易用，非常适合入门，可以带初学者轻松体验 eBPF 的各种跟踪特性。但在上一讲的案例中，你也发现 bpftrace 并不适用于所有的 eBPF 应用，它本身的限制导致我们无法在需要复杂 eBPF 程序的场景中使用它。在复杂的应用中，我还是推荐你使用 BCC 或者 libbpf 进行开发。

那么，今天我就带你看看，如何使用 BCC 和 libbpf 这两个进阶方法来开发内核跟踪程序。

BCC 方法

我们先来看看如何使用 BCC 来开发上一讲中短时进程的跟踪程序。这里先说明下，由于 execveat 的处理逻辑同 execve 基本相同，限于篇幅的长度，接下来的 BCC 和 libbpf 程序都以 execve 为例。

这里我们先回顾下 03 讲 的内容，使用 BCC 开发的 eBPF 程序包含两部分：

> 第一部分是用 C 语言开发的 eBPF 程序。在 eBPF 程序中，你可以利用 BCC 提供的库函数和宏定义简化你的处理逻辑。
>
> 第二部分是用 Python 语言开发的前端界面，其中包含 eBPF 程序加载、挂载到内核函数和跟踪点，以及通过 BPF 映射获取和打印执行结果等部分。在前端程序中，你同样可以利用 BCC 库来访问 BPF 映射。

数据结构定义

我们先看第一部分。为了在系统调用入口跟踪点和出口跟踪点间共享进程信息等数据，我们可以定义一个哈希映射（比如命名为 tasks）；同样地，因为我们想要在用户空间实时获取跟踪信息，这就需要一个性能事件映射。对于这两种映射的创建步骤，BCC 已经提供了非常方便的宏定义，你可以直接使用。

比如，你可以用下面的方式来创建这两个映射：

复制代码

```
struct data_t {
    u32 pid;
    char comm[TASK_COMM_LEN];
    int retval;
    unsigned int args_size;
    char argv[FULL_MAX_ARGS_ARR];
};
BPF_PERF_OUTPUT(events);
BPF_HASH(tasks, u32, struct data_t);
```

代码中指令的具体作用如下：

`struct data_t` 定义了一个包含进程基本信息的数据结构，它将用在哈希映射的值中（其中的参数大小 `args_size` 会在读取参数内容的时候用到）；

`BPF_PERF_OUTPUT(events)` 定义了一个性能事件映射；

`BPF_HASH(tasks, u32, struct data_t)` 定义了一个哈希映射，其键为 32 位的进程 PID，而值则是进程基本信息 `data_t`。

两个映射定义好之后，接下来就是**定义跟踪点的处理函数**。在 BCC 中，你可以通过 `TRACEPOINT_PROBE(category, event)` 来定义一个跟踪点处理函数。BCC 会将所有的参数放入 `args` 这个变量中，这样使用 `args-><参数名>` 就可以访问跟踪点的参数值。

对我们要跟踪的短时进程问题来说，也就是下面这两个跟踪点：

复制代码

```
// 定义sys_enter_execve跟踪点处理函数.
TRACEPOINT_PROBE(syscalls, sys_enter_execve)
{
    //待添加处理逻辑
}

// 定义sys_exit_execve跟踪点处理函数.
TRACEPOINT_PROBE(syscalls, sys_exit_execve)
{
    //待添加处理逻辑
}
```

入口跟踪点处理

对于入口跟踪点 `sys_enter_execve` 的处理，还是按照上一讲中 bpftrace 的逻辑，先获取进程的 PID、进程名称和参数列表之后，再存入刚刚定义的哈希映射中。

其中，进程 PID 和进程名称都比较容易获取。如下面的代码所示，你可以调用 `bpf_get_current_pid_tgid()` 查询进程 PID，调用 `bpf_get_current_comm()` 读取进程名称：

复制代码

```
    // 获取进程PID和进程名称
```

```
    struct data_t data = { };
    u32 pid = bpf_get_current_pid_tgid();  // 取低32位为进程PID
    data.pid = pid;
    bpf_get_current_comm(&data.comm, sizeof(data.comm));
```

而命令行参数的获取就没那么容易了。因为 BCC 把所有参数都放到了 `args` 中，你可以使用 `args->argv` 来访问参数列表：

复制代码

```
const char **argv = (const char **)(args->argv);
```

注意，`argv` 是一个用户空间的字符串数组（指针数组），这就需要调用 `bpf_probe_read` 系列的辅助函数，去这些指针中读取数据。并且，字符串的数量（即参数的个数）和每个字符串的长度（即每个参数的长度）都是未知的，由于 eBPF 栈大小只有 512 字节，如果想要把它们读入一个临时的字符数组中，必须要保证每次读取的内容不超过栈的大小。这类问题有很多种不同的处理方法，其中一个比较简单的方式就是**把多余的参数截断，使用...代替过长的参数。**一般来说，知道了进程的名称和前几个参数，对调试和排错来说就足够了。

你可以定义最大读取的参数个数和参数长度，然后在哈希映射的值中定义一个字符数组，代码如下所示：

复制代码

```
// 定义参数长度和参数个数常量
#define ARGSIZE 64
#define TOTAL_MAX_ARGS 5
#define FULL_MAX_ARGS_ARR (TOTAL_MAX_ARGS * ARGSIZE)

struct data_t {
    ...
    char argv[FULL_MAX_ARGS_ARR];
};
```

有了字符数组，接下来再定义一个辅助函数，从参数数组中读取字符串参数（限定最长 `ARGSIZE`）：

```
// 从用户空间读取字符串
static int __bpf_read_arg_str(struct data_t *data, const char *ptr)
{
    if (data->args_size > LAST_ARG) {
        return -1;
    }

    int ret = bpf_probe_read_user_str(&data->argv[data->args_size], ARGSIZE, (
    if (ret > ARGSIZE || ret < 0) {
        return -1;
    }

    // increase the args size. the first tailing '\0' is not counted and hence
    // would be overwritten by the next call.
    data->args_size += (ret - 1);

    return 0;
}
```

复制代码

在这个函数中，有几点需要你注意：

> bpf_probe_read_user_str() 返回的是包含字符串结束符 \0 的长度。为了拼接所有的字符串，在计算已读取参数长度的时候，需要把 \0 排除在外。
>
> &data->argv[data->args_size] 用来获取要存放参数的位置指针，这是为了把多个参数拼接到一起。
>
> 在调用 bpf_probe_read_user_str() 前后，需要对指针位置和返回值进行校验，这可以帮助 eBPF 验证器获取指针读写的边界（如果你感兴趣，可以参考这篇文章，了解更多的内存访问验证细节）。

有了这个辅助函数之后，因为 eBPF 在老版本内核中并不支持循环（有界循环在 **5.3** 之后才支持），要访问字符串数组，还需要一个小技巧：使用 #pragma unroll 告诉编译器，把源码中的循环自动展开。这就避免了最终的字节码中包含循环。

完整的处理函数如下所示（具体的每一步我都加了详细的注释，你可以参考注释来加深理解）：

```
// 引入内核头文件
#include <uapi/linux/ptrace.h>
#include <linux/sched.h>
```

复制代码

```
#include <linux/fs.h>

// 定义sys_enter_execve跟踪点处理函数.
TRACEPOINT_PROBE(syscalls, sys_enter_execve)
{
    // 变量定义
    unsigned int ret = 0;
    const char **argv = (const char **)(args->argv);

    // 获取进程PID和进程名称
    struct data_t data = { };
    u32 pid = bpf_get_current_pid_tgid();
    data.pid = pid;
    bpf_get_current_comm(&data.comm, sizeof(data.comm));

    // 获取第一个参数（即可执行文件的名字）
    if (__bpf_read_arg_str(&data, (const char *)argv[0]) < 0) {
        goto out;
    }

    // 获取其他参数（限定最多5个）
    #pragma unrollfor (int i = 1; i < TOTAL_MAX_ARGS; i++) {
        if (__bpf_read_arg_str(&data, (const char *)argv[i]) < 0) {
            goto out;
        }
    }

 out:
    // 存储到哈希映射中
    tasks.update(&pid, &data);
    return 0;
}
```

注意，**为了获取内核数据结构的定义，在文件的开头需要引入相关的内核头文件**。此外，读取参数完成之后，不要忘记调用 `tasks.update()` 把进程的基本信息存储到哈希映射中。因为返回值需要等到出口跟踪点时才可以获取，这儿只需要更新哈希映射就可以了，不需要把进程信息提交到性能事件映射中去。

出口跟踪点处理

入口跟踪点 `sys_enter_execve` 处理好之后，我们再来看看出口跟踪点 `sys_exit_execve` 该如何处理。

由于进程的基本信息已经保存在了哈希映射中，所以出口事件的处理可以分为查询进程基本信息、填充返回值、最后再提交到性能事件映射这三个步骤。具体代码如下所示：

复制代码

```
// 定义sys_exit_execve跟踪点处理函数.
TRACEPOINT_PROBE(syscalls, sys_exit_execve)
{
    // 从哈希映射中查询进程基本信息
    u32 pid = bpf_get_current_pid_tgid();
    struct data_t *data = tasks.lookup(&pid);

    // 填充返回值并提交到性能事件映射中
    if (data != NULL) {
        data->retval = args->ret;
        events.perf_submit(args, data, sizeof(struct data_t));

        // 最后清理进程信息
        tasks.delete(&pid);
    }

    return 0;
}
```

到这里，完整的 eBPF 程序就开发好了，你可以把上述的代码保存到一个本地文件中，并命名为 `execsnoop.c`（你也可以在 GitHub 上找到全部源码）。

Python 前端处理

eBPF 程序开发完成后，最后一步就是为它增加一个 Python 前端。

同 03 讲 的 Hello World 类似，**Python 前端逻辑需要 eBPF 程序加载、挂载到内核函数和跟踪点，以及通过 BPF 映射获取和打印执行结果等几个步骤**。其中，因为我们已经使用了 `TRACEPOINT_PROBE` 宏定义，来定义 eBPF 跟踪点处理函数，BCC 在加载字节码的时候，会帮你自动把它挂载到正确的跟踪点上，所以挂载的步骤就可以忽略。完整的 Python 程序如下所示：

复制代码

```
# 引入库函数
from bcc import BPF
from bcc.utils import printb

# 1) 加载eBPF代码
b = BPF(src_file="execsnoop.c")

# 2) 输出头
print("%-6s %-16s %-3s %s" % ("PID", "COMM", "RET", "ARGS"))
```

```
10
11 # 3) 定义性能事件打印函数
12 def print_event(cpu, data, size):
13     # BCC自动根据"struct data_t"生成数据结构
14     event = b["events"].event(data)
15     printb(b"%-6d %-16s %-3d %-16s" % (event.pid, event.comm, event.retval, ev
16
17 # 4) 绑定性能事件映射和输出函数，并从映射中循环读取数据
18 b["events"].open_perf_buffer(print_event)
19 while 1:
20     try:
21         b.perf_buffer_poll()
22     except KeyboardInterrupt:
23         exit()
```

把上述的代码保存到 `execsnoop.py` 中，然后通过 Python 运行，并在另一个终端中执行 `ls` 命令，你就可以得到如下的输出：

复制代码

```
1 $ sudo python3 execsnoop.py
2 PID    COMM             RET ARGS
3 249134 zsh              0   ls--color=tty
```

恭喜，到这里你已经开发了一个新的 eBPF 程序，并且它可以帮你排查短时进程相关的性能问题。

不过，在你想要分发这个程序到生产环境时，又会碰到一个新的难题：BCC 依赖于 LLVM 和内核头文件才可以动态编译和加载 eBPF 程序，而出于安全策略的需要，在生产环境中通常又不允许安装这些开发工具。

这个难题应该怎么克服呢？一种很容易想到的方法是把 BCC 和开发工具都安装到容器中，容器本身不提供对外服务，这样可以降低安全风险。另外一种方法就是参考内核中的 eBPF 示例，开发一个匹配当前内核版本的 eBPF 程序，并编译为字节码，再分发到生产环境中。

除此之外，如果你的内核已经支持了 BPF 类型格式 (BTF)，我推荐你使用从内核源码中抽离出来的 libbpf 进行开发，这样可以借助 BTF 和 CO-RE 获得更好的移植性。实际上，

BCC 的很多工具都在向 BTF 迁移中，相信未来 libbpf 会成为最受欢迎的 eBPF 程序开发基础库，甚至 Windows eBPF 也会支持 libbpf。

libbpf 方法

那么，如何用 libbpf 来开发一个 eBPF 程序呢？跟刚才的 BCC 程序类似，使用 libbpf 开发 eBPF 程序也是分为两部分：第一，内核态的 eBPF 程序；第二，用户态的加载、挂载、映射读取以及输出程序等。

在 eBPF 程序中，由于内核已经支持了 BTF，你不再需要引入众多的内核头文件来获取内核数据结构的定义。取而代之的是一个通过 bpftool 生成的 `vmlinux.h` 头文件，其中包含了内核数据结构的定义。

这样，使用 libbpf 开发 eBPF 程序就可以通过以下四个步骤完成：

1. 使用 bpftool 生成内核数据结构定义头文件。BTF 开启后，你可以在系统中找到 `/sys/kernel/btf/vmlinux` 这个文件，bpftool 正是从它生成了内核数据结构头文件。
2. 开发 eBPF 程序部分。为了方便后续通过统一的 Makefile 编译，eBPF 程序的源码文件一般命名为 `<程序名>.bpf.c`。
3. 编译 eBPF 程序为字节码，然后再调用 `bpftool gen skeleton` 为 eBPF 字节码生成脚手架头文件（Skeleton Header）。这个头文件包含了 eBPF 字节码以及相关的加载、挂载和卸载函数，可在用户态程序中直接调用。
4. 最后就是用户态程序引入上一步生成的头文件，开发用户态程序，包括 eBPF 程序加载、挂载到内核函数和跟踪点，以及通过 BPF 映射获取和打印执行结果等。

通常，这几个步骤里面的编译、库链接、执行 `bpftool` 命令等，都可以放到 Makefile 中，这样就可以通过一个 `make` 命令去执行所有的步骤。比如，下面是一个简化版本的 Makefile：

复制代码

```
APPS = execsnoop

.PHONY: all
all: $(APPS)
```

```
$(APPS):
    clang -g -O2 -target bpf -D__TARGET_ARCH_x86_64 -I/usr/include/x86_64-linu
    bpftool gen skeleton $@.bpf.o > $@.skel.h
    clang -g -O2 -Wall -I . -c $@.c -o $@.o
    clang -Wall -O2 -g $@.o -static -lbpf -lelf -lz -o $@

vmlinux:
    $(bpftool) btf dump file /sys/kernel/btf/vmlinux format c > vmlinux.h

```

有了这个 Makefile 之后，你执行 `make vmlinux` 命令就可以生成 `vmlinux.h` 文件，再执行 `make` 就可以编译 `APPS` 里面配置的所有 eBPF 程序（多个程序之间以空格分隔）。

接下来，我就带你一起通过上述四个步骤开发跟踪短时进程的 eBPF 程序。

内核头文件生成

首先，对于第一步，我们只需要执行下面的命令，即可生成内核数据结构的头文件：

复制代码

```
sudo bpftool btf dump file /sys/kernel/btf/vmlinux format c > vmlinux.h
```

如果命令执行失败了，并且错误说 BTF 不存在，那说明当前系统内核没有开启 BTF 特性。这时候，你需要开启 `CONFIG_DEBUG_INFO_BTF=y` 和 `CONFIG_DEBUG_INFO=y` 这两个编译选项，然后重新编译和安装内核。

eBPF 程序定义

第二步就是开发 eBPF 程序，包括定义哈希映射、性能事件映射以及跟踪点的处理函数等，而对这些数据结构和跟踪函数的定义都可以通过 `SEC()` 宏定义来完成。在编译时，**通过 `SEC()` 宏定义的数据结构和函数会放到特定的 ELF 段中，这样后续在加载 BPF 字节码时，就可以从这些段中获取所需的元数据。**

比如，你可以使用下面的代码来定义映射和跟踪点处理函数：

复制代码

```
// 包含头文件
#include "vmlinux.h"
#include <bpf/bpf_helpers.h>

// 定义进程基本信息数据结构
struct event {
    char comm[TASK_COMM_LEN];
    pid_t pid;
    int retval;
    int args_count;
    unsigned int args_size;
    char args[FULL_MAX_ARGS_ARR];
};

// 定义哈希映射
struct {
    __uint(type, BPF_MAP_TYPE_HASH);
    __uint(max_entries, 10240);
    __type(key, pid_t);
    __type(value, struct event);
} execs SEC(".maps");

// 定义性能事件映射
struct {
    __uint(type, BPF_MAP_TYPE_PERF_EVENT_ARRAY);
    __uint(key_size, sizeof(u32));
    __uint(value_size, sizeof(u32));
} events SEC(".maps");

// sys_enter_execve跟踪点
SEC("tracepoint/syscalls/sys_enter_execve")
int tracepoint__syscalls__sys_enter_execve(struct trace_event_raw_sys_enter *c
{
  // 待实现处理逻辑
}

// sys_exit_execve跟踪点
SEC("tracepoint/syscalls/sys_exit_execve")
int tracepoint__syscalls__sys_exit_execve(struct trace_event_raw_sys_exit *ctx
{
  // 待实现处理逻辑
}

// 定义许可证（前述的BCC默认使用GPL）
char LICENSE[] SEC("license") = "Dual BSD/GPL";
```

让我们来看看这段代码的具体含义：

头文件 vmlinux.h 包含了内核数据结构，而 bpf/bpf_helpers.h 包含了 05 讲 提到的 BPF 辅助函数；

struct event 定义了进程基本信息数据结构，它会用在后面的哈希映射中；

SEC(".maps") 定义了哈希映射和性能事件映射；

SEC("tracepoint/<跟踪点名称>") 定义了跟踪点处理函数，系统调用跟踪点的格式是 tracepoint/syscalls/<系统调用名称>"。以后你需要定义内核插桩和用户插桩的时候，也是以类似的格式定义，比如 kprobe/do_unlinkat 或 uprobe/func；

最后的 SEC("license") 定义了 eBPF 程序的许可证。在上述的 BCC eBPF 程序中，我们并没有定义许可证，这是因为 BCC 自动帮你使用了 GPL 许可。

有了基本的程序结构，接下来就是**实现系统调用入口和出口跟踪点的处理函数**。它们的基本过程跟上述的 BCC 程序是类似的。

入口跟踪点处理

对于入口跟踪点 sys_enter_execve 的处理，还是按照上述 BCC 程序的逻辑，先获取进程的 PID、进程名称和参数列表之后，再存入刚刚定义的哈希映射中。完整代码如下所示，具体每一步的内容我都加了详细的注释：

复制代码

```
SEC("tracepoint/syscalls/sys_enter_execve")
int tracepoint__syscalls__sys_enter_execve(struct trace_event_raw_sys_enter
                        *ctx)
{
    struct event *event;
    const char **args = (const char **)(ctx->args[1]);
    const char *argp;

    // 查询PID
    u64 id = bpf_get_current_pid_tgid();
    pid_t pid = (pid_t) id;

    // 保存一个空的event到哈希映射中
    if (bpf_map_update_elem(&execs, &pid, &empty_event, BPF_NOEXIST)) {
        return 0;
    }
    event = bpf_map_lookup_elem(&execs, &pid);
    if (!event) {
        return 0;
    }
```

```
    // 初始化event变量
    event->pid = pid;
    event->args_count = 0;
    event->args_size = 0;

    // 查询第一个参数
    unsigned int ret = bpf_probe_read_user_str(event->args, ARGSIZE,
                            (const char *)ctx->args[0]);
    if (ret <= ARGSIZE) {
        event->args_size += ret;
    }

    // 查询其他参数
    event->args_count++;
    #pragma unrollfor (int i = 1; i < TOTAL_MAX_ARGS; i++) {
        bpf_probe_read_user(&argp, sizeof(argp), &args[i]);
        if (!argp)
            return 0;

        if (event->args_size > LAST_ARG)
            return 0;

        ret =
            bpf_probe_read_user_str(&event->args[event->args_size],
                        ARGSIZE, argp);
        if (ret > ARGSIZE)
            return 0;

        event->args_count++;
        event->args_size += ret;
    }

    // 再尝试一次，确认是否还有未读取的参数
    bpf_probe_read_user(&argp, sizeof(argp), &args[TOTAL_MAX_ARGS]);
    if (!argp)
        return 0;

    // 如果还有未读取参数，则增加参数数量（用于输出"..."）
    event->args_count++;

    return 0;
}
```

其中，你需要注意这三点：

第一，程序使用了 bpf_probe_read_user() 来查询参数。由于它把 \0 也算到了已读取参数的长度里面，所以最终 event->args 中保存的各个参数是以 \0 分隔

的。在用户态程序输出参数之前，需要用空格替换 \0。

第二，程序在一开始的时候向哈希映射存入了一个空事件，在后续出口跟踪点处理的时候需要确保空事件也能正确清理。

第三，程序在最后又尝试多读取了一次参数列表。如果还有未读取参数，参数数量增加了 1。用户态程序可以根据参数数量来决定是不是需要在参数结尾输出一个 ...。

出口跟踪点处理

入口跟踪点处理好之后，再来看看出口跟踪点的处理方法。它的步骤跟 BCC 程序也是类似的，也是查询进程基本信息、填充返回值、提交到性能事件映射这三个步骤。

除此之外，由于刚才入口跟踪点的处理中没有读取进程名称，所以在提交性能事件之前还需要先查询一下进程名称。完整的程序如下所示，具体每一步的内容我也加了详细的注释：

复制代码

```
SEC("tracepoint/syscalls/sys_exit_execve")
int tracepoint__syscalls__sys_exit_execve(struct trace_event_raw_sys_exit *ctx
{
    u64 id;
    pid_t pid;
    int ret;
    struct event *event;

    // 从哈希映射中查询进程基本信息
    id = bpf_get_current_pid_tgid();
    pid = (pid_t) id;
    event = bpf_map_lookup_elem(&execs, &pid);
    if (!event)
        return 0;

    // 更新返回值和进程名称
    ret = ctx->ret;
    event->retval = ret;
    bpf_get_current_comm(&event->comm, sizeof(event->comm));

    // 提交性能事件
    size_t len = EVENT_SIZE(event);
    if (len <= sizeof(*event))
        bpf_perf_event_output(ctx, &events, BPF_F_CURRENT_CPU, event,
                      len);

    // 清理哈希映射
```

```
28     bpf_map_delete_elem(&execs, &pid);
29     return 0;
30 }
```

从这些代码中你可以看到，它的处理逻辑跟上述的 BCC 程序基本上是相同的。不过，详细对比一下，你会发现它们之间还是有不同的，不同点主要在两个方面：

第一，函数名的定义格式不同。BCC 程序使用的是 `TRACEPOINT_PROBE` 宏，而 libbpf 程序用的则是 `SEC` 宏。

第二，映射的访问方法不同。BCC 封装了很多更易用的映射访问函数（如 `tasks.lookup()`），而 libbpf 程序则需要调用 05 讲 提到过的 BPF 辅助函数（比如查询要使用 `bpf_map_lookup_elem()`）。

到这里，新建一个目录，并把上述代码存入 `execsnoop.bpf.c` 文件中，eBPF 部分的代码也就开发好了。

编译并生成脚手架头文件

有了 eBPF 程序，执行下面的命令，你就可以使用 clang 和 bpftool 将其编译成 BPF 字节码，然后再生成其脚手架头文件 `execsnoop.skel.h`（注意，脚手架头文件的名字一般定义为 `<程序名>.skel.h`）：

复制代码

```
clang -g -O2 -target bpf -D__TARGET_ARCH_x86_64 -I/usr/include/x86_64-linux-gn
bpftool gen skeleton execsnoop.bpf.o > execsnoop.skel.h
```

其中，clang 的参数 `-target bpf` 表示要生成 BPF 字节码，`-D__TARGET_ARCH_x86_64` 表示目标的体系结构是 x86_64，而 `-I` 则是引入头文件路径。

命令执行后，脚手架头文件会放到 `execsnoop.skel.h` 中，这个头文件包含了 BPF 字节码和相关的管理函数。因而，当用户态程序引入这个头文件并编译之后，只需要分发最终用户态程序生成的二进制文件到生产环境即可（如果用户态程序使用了其他的动态库，还需要分发动态库）。

开发用户态程序

有了脚手架头文件之后，还剩下最后一步，也就是用户态程序的开发。

同 BCC 的 Python 前端程序类似，libbpf 用户态程序也需要 eBPF 程序加载、挂载到跟踪点，以及通过 BPF 映射获取和打印执行结果等几个步骤。虽然 C 语言听起来可能比 Python 语言麻烦一些，但实际上，这几个步骤都可以通过脚手架头文件中自动生成的函数来完成。

下面是忽略了错误处理逻辑之后，用户态程序的一个基本框架：

复制代码

```c
// 引入脚手架头文件
#include "execsnoop.skel.h"

// C语言主函数
int main(int argc, char **argv)
{
    // 定义BPF程序和性能事件缓冲区
    struct execsnoop_bpf *skel;
    struct perf_buffer_opts pb_opts;
    struct perf_buffer *pb = NULL;
    int err;

    // 1. 设置调试输出函数
    libbpf_set_print(libbpf_print_fn);

    // 2. 增大 RLIMIT_MEMLOCK（默认值通常太小，不足以存入BPF映射的内容）
    bump_memlock_rlimit();

    // 3. 初始化BPF程序
    skel = execsnoop_bpf__open();

    // 4. 加载BPF字节码
    err = execsnoop_bpf__load(skel);

    // 5. 挂载BPF字节码到跟踪点
    err = execsnoop_bpf__attach(skel);

    // 6. 配置性能事件回调函数
    pb_opts.sample_cb = handle_event;
    pb = perf_buffer__new(bpf_map__fd(skel->maps.events), 64, &pb_opts);

    // 7. 从缓冲区中循环读取数据
    while ((err = perf_buffer__poll(pb, 100)) >= 0) ;
}
```

其中，execsnoop_ 开头的数据结构和函数都包含在脚手架头文件 execsnoop.skel.h 中。而具体到每一步的含义如下：

第 1 步的调试输出函数中，可以调用 printf() 把调试信息输出到终端中。

第 2 步增大锁定内存限制 RLIMIT_MEMLOCK 是必要的，因为系统默认的锁定内存通常过小，无法满足 BPF 映射的需要。

第 3~5 步，直接调用脚手架头文件中的函数，加载 BPF 字节码并挂载到跟踪点。

第 6~7 步为性能事件设置回调函数，并从缓冲区中循环读取数据。注意，性能事件映射 skel->maps.events 也是 bpftool 自动帮你生成好的。

接下来，在性能事件回调函数中，把数据格式转换为 struct event 格式之后，由于参数列表是使用 \0 来分割的，并不能直接向终端打印所有参数。所以，还需要把 \0 先替换为空格，然后再打印。完整的回调函数如下所示：

复制代码

```
// 性能事件回调函数(向终端中打印进程名、PID、返回值以及参数)
void handle_event(void *ctx, int cpu, void *data, __u32 data_sz)
{
    const struct event *e = data;
    printf("%-16s %-6d %3d ", e->comm, e->pid, e->retval);
    print_args(e);
    putchar('\n');
}

// 打印参数(替换'\0'为空格)
static void print_args(const struct event *e)
{
    int args_counter = 0;

    for (int i = 0; i < e->args_size && args_counter < e->args_count; i++) {
        char c = e->args[i];
        if (c == '\0') {
            // 把'\0'替换为空格
            args_counter++;
            putchar(' ');
        } else {
            putchar(c);
        }
    }
    if (e->args_count > TOTAL_MAX_ARGS) {
        // 过长的参数输出"..."替代
```

```
27          fputs(" ...", stdout);
28      }
29  }
```

把上面的代码保存到 execsnoop.c 文件中（你也可以在 GitHub 上找到完整的代码），然后执行下面的命令，将其编译为可执行文件：

复制代码

```
1 clang -g -O2 -Wall -I . -c execsnoop.c -o execsnoop.o
2 clang -Wall -O2 -g execsnoop.o -static -lbpf -lelf -lz -o execsnoop
```

最后，执行 execsnoop，你就可以得到如下的结果：

复制代码

```
1 $ sudo ./execsnoop
2 COMM             PID    RET ARGS
3 sh               276871   0 /bin/sh -c which ps
4 which            276872   0 /usr/bin/which ps
```

你还可以直接把这个文件复制到开启了 BTF 的其他机器中，无需安装额外的 LLVM 开发工具和内核头文件，也可以直接执行。

如果命令失败，并且你看到如下的错误，这说明当前机器没有开启 BTF，需要重新编译内核开启 BTF 才可以运行：

复制代码

```
1 Failed to load and verify BPF skeleton
```

恭喜，加上上一讲的内容，到这里你就通过 bpftrace、BCC 和 libbpf 这三种方法，实现了短时进程的跟踪。虽然这三种方法的步骤和实现代码各不相同，但实际上它们的实现逻辑都是类似的，无非就是**找出跟踪点，然后在 eBPF 部分获取想要的数据并保存到 BPF 映射中，最后在用户空间程序中读取 BPF 映射的内容并输出出来**。

小结

今天，我以短时进程的跟踪为例，通过 BCC 和 libbpf 这两种方法实现了短时进程的跟踪程序（你可以在 GitHub 的这个链接中，找到今天的案例中提到的所有源码）。加上上一讲介绍的 bpftrace 方法，我已经带你掌握了目前最常用的三种 eBPF 程序开发方法，在这里我一起总结下。

在实际的应用中，这三种方法有不同的使用场景：

- bpftrace 通常用在**快速排查和定位系统**上，它支持用单行脚本的方式来快速开发并执行一个 eBPF 程序；
- BCC 通常用在**开发复杂的 eBPF 程序**中，它内置的各种小工具也是目前应用最为广泛的 eBPF 小程序；
- libbpf 是**从内核中抽离出来的标准库**，用它开发的 eBPF 程序可以直接分发执行，不再需要在每台机器上都安装 LLVM 和内核头文件。

通常情况下，你可以用 bpftrace 或 BCC 做一些快速原型，验证你的设计思路是不是可行，然后再切换到 libbpf ，开发完善的 eBPF 程序后再去分发执行。这样，不仅 eBPF 程序运行得更快（无需编译步骤），还避免了在运行环境中安装开发工具和内核头文件。

在不支持 BTF 的机器中，如果不想在运行 eBPF 时依赖于 LLVM 编译和内核头文件，你还可以参考内核中的 BPF 示例，直接引用内核源码中的 `tools/lib/bpf/` 库，以及内核头文件中的数据结构，来开发 eBPF 程序。

思考题

今天使用 BCC 和 libbpf 开发的 eBPF 程序虽然可以正常运行，但是我相信细心的你一定发现了，它还有不少小问题，比如：

- 单个参数过长，或者总的参数数量比较多时，都会被截断，没法完整显示所有的参数列表；
- 在调试短时进程问题时，很多情况下我们可能还需要父进程的信息，这样才能更快定位它们都是被哪些进程创建出来的。

学习完最近这两讲的内容，你觉得该如何解决这些问题呢？你可以在 execsnoop 的基础上改进，开发一个更完善的 eBPF 程序吗？欢迎在评论区和我分享你的思路和解决方

法。

期待你在留言区和我讨论，也欢迎把这节课分享给你的同事、朋友。让我们一起在实战中演练，在交流中进步。

分享给需要的人，Ta订阅超级会员，你将得 50 元

Ta单独购买本课程，你将得 20 元

生成海报并分享

赞 2 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 07 | 内核跟踪（上）：如何查询内核中的跟踪点？

下一篇 09 | 用户态跟踪：如何使用 eBPF 排查应用程序？

精选留言 (4)

写留言

莫名

2022-02-02

1、对于参数问题，当前 BPF 程序尝试把所有参数一次性放入，受限于栈最大长度 512，很容易出现被截断现象。以 BCC 程序为例的解决方法：遍历参数列表 argv 时，每个参数读取之后直接调用 perf_submit 提交至 ringbuf，而不是读取所有参数后仅提交一次，最后用户态程序负责把这些字符串拼接起来。这样可以做到参数最大个数不受限制，且每个参数长度可接近栈最大长度 512（当前 BPF 程序限制 64 容易被截断）。...

展开

作者回复: 非常棒的答案，赞一个

共 2 条评论 > 2

c1

2022-02-09

请教：
Ubuntu 21.10
Linux u21 5.13.0-28-generic #31-Ubuntu SMP Thu Jan 13 17:41:06 UTC 2022 x86_64 x86_64 x86_64 GNU/Linux
...
展开

作者回复: 有没有引入头文件？完整的程序可以参考Github： https://github.com/feiskyer/ebpf-apps/blob/main/bcc-apps/python/execsnoop.c#L2-L4。

写点啥呢
2022-02-03

请教下老师：
1. bpf的辅助函数执行上下文是在当前进程下么，因为我看像execsnoop例子中get_current_pid_tgid调用能拿到的是新启动进程pid。
2. libbpf的开发模式能看到从编译器到bpf在背后做了很多工作，如果想了解bpf程序的实现，比如宏是如何定义了映射和挂载点，程序加载的时候如何从程序段中的信息实现的...
展开

作者回复: 1. eBPF程序是在内核空间执行，可以在内核空间获得进程的上下文信息。
2. 这些具体的步骤可以参考 https://github.com/libbpf/libbpf。比如，https://github.com/libbpf/libbpf/blob/master/src/libbpf.c#L8599-L8675 这儿定义了所有不同程序类型所对应的 SEC 名称及挂载方式。

共 2 条评论 >

Haric
2022-02-03

文章提到，使用libbfp方式在开启了 BTF 的其他机器都可以运行，请问在嵌入式设备（需要交叉编译）能运行吗？

作者回复: 如果内核也支持BTF的话，理论上应该是可以运行的；但如果内核版本比较旧的话，可能不支持BTF，这时候可以使用条件编译的方式，为旧版本内核关闭BTF。